Chapter 13

VERIFICATION OF A SIMPLE PIPELINED MACHINE MODEL

Jun Sawada

University of Texas at Austin

sawada@cs.utexas.edu

Abstract The difficulty of pipelined machine verification derives from the fact that there is a complex time-abstraction between the pipelined implementation and its specification that executes instructions sequentially. To study this problem, we define a simple three-stage pipelined machine in ACL2. We prove that this pipelined machine returns the same result as its specification machine. In order to ease the proof, we define an intermediate abstraction called MAETT. This abstraction models the behavior of instructions in the pipelined architecture, and it allows us to define directly and verify invariant conditions about executed instructions. The author used a similar approach to verify a more realistic pipelined machine. This section serves as an introduction to the verification of pipelined machines.

1. A THREE-STAGE PIPELINED MACHINE AND ITS CORRECTNESS

Pipelining is a key idea in the design of modern microprocessors. It improves the performance of microprocessors by overlapping the execution of instructions. A pipelined microprocessor typically starts the execution of an instruction before the completion of the previous instruction. However, programmers imagine that microprocessors execute instructions one-by-one. This makes it natural to define the specification of a microprocessor as a sequential execution model. Therefore, the verification of a pipelined microprocessor needs to prove that the pipelined implementation appears to behave as its sequential execution model does.

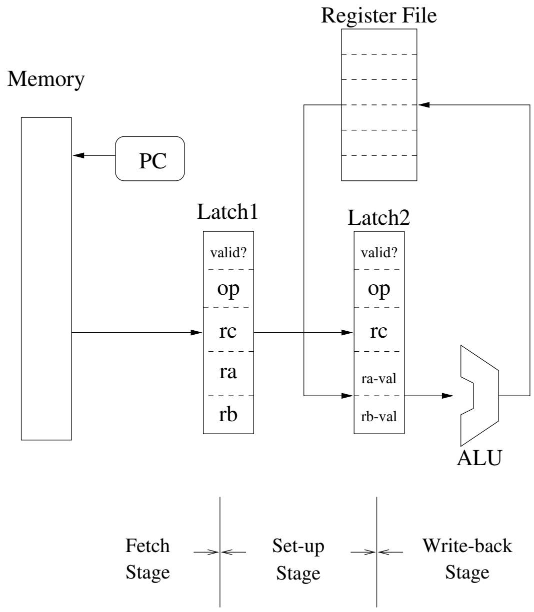

Figure 13.1 The three-stage pipelined machine.

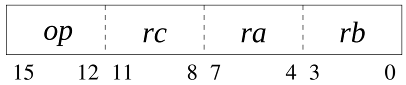

Figure 13.2 The instruction format for the three-stage pipelined machine.

To study this problem, we will consider a three-stage pipelined machine. Figure 13.1 shows its block diagram. This machine consists of a program counter (PC), a register file, a memory, an ALU, and two pipeline latches. The register file is a collection of registers.

The instruction format for this machine is shown in Fig. 13.2. An instruction is a 16-bit word and it has four fields: opcode field *op*, destination register field *rc*, source register fields *ra* and *rb*. For example, the bits between the 12th bit and the 15th bit are the opcode, which specifies the instruction type. The opcodes of ADD and SUB instructions are 0 and 1, respectively. An instruction with opcode other than 0 and 1 is considered to be a NOP, which only increments the program counter.

There are three stages in the pipeline: the fetch stage, the set-up stage, and the write-back stage. The machine fetches an instruction in the fetch stage, reads source registers in the set-up stage, and performs an arithmetic operation and updates the destination register in the write-back stage. The latches are used to store intermediate results. The *valid?* flag of each latch is set to 1 when an instruction occupies the latch. The *op* field stores the opcode of the stored instruction, and the *rc*, *ra*, and *rb* fields store the operand register designators. The *ra-val* and *rb-val* fields store the two source operand register values.

Let us consider the execution of the following three instructions. In this program, the operand registers *rc*, *ra* and *rb* are printed in that order. For example, i_0 is an ADD instruction that reads registers R1 and R3 and stores the results in R2.

i_0: ADD R2, R1, R3
i_1: SUB R4, R2, R5
i_2: ADD R7, R5, R6

Table 13.1 shows the latches in which the intermediate results of instructions are stored at each time. For example, the instruction i_0 is fetched between time 0 and 1, it goes through the set-up stage between time 1 and 2, and it finishes the write-back stage between time 2 and 3. Thus, the intermediate results of i_0 are stored in latch1 at time 1 and in latch2 at time 2.

Table 13.1 A reservation table for the three-stage pipelined machine.

Time	0	1	2	3	4	5	6
i_0		latch1	latch2				
i_1			latch1	latch1	latch2		
i_2					latch1	latch2	

This table shows a typical pipelined execution. While i_0 is at the set-up stage between time 1 and 2, instruction i_1 is fetched simultaneously. Since the instruction i_1 uses the value of register R2 which is the result of instruction i_0, instruction i_1 must wait for i_0 to update R2 before reading its value in the set-up stage. Thus, the instruction i_1 *stalls* between times 2 and 3; i_1 stays in latch1. After i_0 completes the write-back stage at time 3, i_1 continues the rest of the execution in the set-up and write-back stages. Instruction i_2 is executed between times 3 and 6.

Because the execution of instructions is overlapped, a pipelined machine state may not correspond to any state which programmers expect to see. For example at time 3, the program counter points to the next instruction to be fetched, namely i_2. However, the register file records the result of i_0, but not i_1 yet, since i_1 is still at latch1. In other words, the program counter appears as if we have completed two instructions i_0 and i_1, but the registers appear as if we have only completed the instruction i_0. Thus, the pipeline state at time 3 does not correspond to any state observable by executing instructions sequentially.

To be more concrete, we define the machine in the ACL2 logic at two levels: the *instruction-set architecture* (ISA) level and the *microarchitecture* (MA) level. The ISA models the machine behavior that programmers have in mind. It executes instructions one at a time. This style of execution is called *sequential execution*. On the other hand, the MA model defines how the actual pipelined machine behaves. Instruction executions are overlapped in this model, and this style of execution is called *pipelined execution*.

The behavior of the ISA model is given by the following function.

```
(defun ISA-step (ISA)
  (let ((inst (read-mem (ISA-pc ISA) (ISA-mem ISA))))
    (let ((op (op-field inst))
          (rc (rc-field inst))
          (ra (ra-field inst))
          (rb (rb-field inst)))
      (cond ((equal op 0)              ; ADD
             (ISA-add rc ra rb ISA))
            ((equal op 1)              ; SUB
             (ISA-sub rc ra rb ISA))
            (t (ISA-default ISA))))))  ; NOP
```

This function takes the current state `ISA` and returns the new state after executing one instruction. A state of the ISA model is represented with an ACL2 structure defined by `defstructure`, which will be discussed later. Expressions `(ISA-pc ISA)`, `(ISA-regs ISA)`, and `(ISA-mem ISA)` return the program counter, the register file and the memory in state `ISA`, respectively. `ISA-step` reads an instruction `inst` from the memory at the location addressed by the program counter, divides it into instruction fields, and executes it appropriately depending on the opcode. For example, if the `op-field` of `inst` contains 0, the `ISA-step` function performs an ADD instruction whose effect is defined by the function `ISA-add`. Similarly, `ISA-sub` and `ISA-default` define the effects of the SUB and NOP instructions, respectively.

We can define a recursive function `(ISA-stepn ISA n)`, which calculates the result of executing `n` instructions. It is defined to apply `ISA-step` repeatedly `n` times.

```
(defun ISA-stepn (ISA n)
  (if (zp n) ISA (ISA-stepn (ISA-step ISA) (1- n))))
```

The expression `(MA-step MA sig)` takes the current pipeline state `MA` and an external input signal `sig`, and returns the pipeline state at the next clock cycle. It defines the behavior of the pipelined machine at the MA level, by specifying how individual components behave in every clock cycle.

```
(defun MA-step (MA sig)
  (MA-state (step-pc MA sig)
            (step-regs MA)
            (MA-mem MA)
            (step-latch1 MA sig)
            (step-latch2 MA)))
```

Functions `step-pc`, `step-regs`, `step-latch1`, and `step-latch2` define the new state of the program counter, the register file, and pipeline latches latch1 and latch2. The memory state does not change in this simple machine. Constructor function `MA-state` combines these component states to form the new MA state. The recursive function `(MA-stepn MA sig-list n)` applies `MA-step` repeatedly `n` times and returns the MA state `n` clock cycles later. The argument `sig-list` is a list of input signals.

```
(defun MA-stepn (MA sig-list n)
  (if (zp n)
      MA
      (MA-stepn (MA-step MA (car sig-list))
                (cdr sig-list)
                (1- n))))
```

One key idea in comparing pipelined machine states to sequential execution states is using *pipeline flushed states.* An MA state is a pipeline flushed state if no instructions are partially executed in the pipeline. In Table 13.1, the MA is in pipeline flushed states at time 0 and 6. In a pipeline flushed state, all programmer visible components, such as the program counter, the register file, and the memory, are synchronized. Thus it is easy to define the corresponding ISA state for a pipeline flushed state. We define this correspondence as a projection function `(proj MA)`, which returns an ISA state by extracting the program counter, the register file, and the memory states from the pipeline state `MA`.

```
(defun proj (MA)
  (ISA-state (MA-pc MA) (MA-regs MA) (MA-mem MA)))
```

Figure 13.3 shows the commutative diagram that represents the correctness of our pipelined machine. Consider an initial state MA_0, which

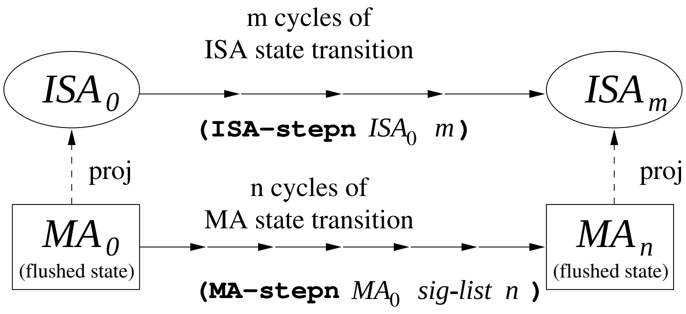

Figure 13.3 The commutative diagram for the pipelined machine.

we assume is a pipeline flushed state. There are two paths to follow in the commutative diagram. One path runs the MA model for n steps where n is an arbitrary natural number. Suppose the final state MA_n is also a flushed state. Then we can map the final state MA_n to ISA_m with the projection function. Let m be the number of instructions executed during the state transition from MA_0 to MA_n. The other path first projects the initial state MA_0 to ISA_0 and runs the ISA for m cycles to reach ISA_m. If the MA correctly implements the ISA, the same ISA_m must be obtained by following the two paths. We will prove this commutative diagram in the rest of the chapter.

2. INTERMEDIATE ABSTRACTION AND INVARIANT

It is often difficult to directly verify an entire pipelined machine. Our example machine is simple, but a typical pipelined microprocessor has a long pipeline with a complex control logic. Instead of directly analyzing the entire microarchitecture, we show that each instruction is executed correctly. This allows us to verify the machine design incrementally, and later combine the results together to prove the commutative diagram.

To pursue this idea, our verification approach first defines an intermediate abstraction, which builds a list of completely executed instructions and in-flight instructions. For example at time 4 in Table 13.1, instruction i_0 has been completely executed, and i_1 and i_2 are in-flight. The intermediate abstraction represents the MA state at time 4 with a list of instructions i_0, i_1 and i_2.

More precisely speaking, the status of each instruction is recorded in the intermediate abstraction. We represent the status of an instruction with a structure type named `INST`. In ACL2, the structure can be defined with `defstructure`[1] macro as follows:

```
(defstructure INST
  (stg      (:assert (stage-p stg)          :rewrite))
  (pre-ISA  (:assert (ISA-state-p pre-ISA)  :rewrite))
  (post-ISA (:assert (ISA-state-p post-ISA) :rewrite))
  (:options ...)) .
```

This structure has three fields `stg`, `pre-ISA`, and `post-ISA`. Field `stg` represents the current stage of the represented instruction, and `pre-ISA` and `post-ISA` store ISA states which we will describe shortly. Accessors functions to the structure fields are `INST-stg`, `INST-pre-ISA`, and `INST-post-ISA`.

Let i_k^t denote the `INST` structure representing the status of instruction i_k at time t in Table 13.1. Since i_0 is at latch1 at time 1, `(INST-stg` i_0^1`)` = `'latch1`. Similarly, `(INST-stg` i_0^2`)` = `'latch2`. The stage of completed instructions is defined as `'retire`, so `(INST-stg` i_0^3`)` = `'retire`.

Using this instruction representation, we define the intermediate abstraction state. We call this intermediate abstraction a *Microarchitecture Execution Trace Table* (MAETT)[Sawada and Hunt, 1997]. It is defined using the ACL2 structure:

```
(defstructure MAETT
  (init-ISA (:assert (ISA-state-p init-ISA) :rewrite))
  (trace    (:assert (INST-listp trace)     :rewrite))
  (:options (:conc-name MT-) ...)) .
```

The `trace` field stores the list of completed and in-flight instructions. Let MT_t be the MAETT for the MA state at time t in Table 13.1. The `trace` field of initial MAETT MT_0 contains an empty list `nil`. As more instructions are fetched, the MAETT adds to the list INST items which represent the fetched instructions. For example, the `trace` field of MT_1 and MT_2 store list (i_0^1) and $(i_0^2\ i_1^2)$, respectively.

The `init-ISA` field of a MAETT stores the initial ISA state before the execution of the first instruction in the program. Additionally, the `pre-ISA` and `post-ISA` fields of the `INST` structure store the ISA state before and after executing the represented instruction in the ISA model. We call these states the *pre-ISA state* and the *post-ISA state* of the instruction. Figure 13.4 shows the entire structure of the MAETT MT_4. The `trace` field stores the list $(i_0^4\ i_1^4\ i_2^4)$. The `init-ISA` field stores the initial ISA state ISA_0. This is also the pre-ISA state of the first instruction i_0. The result of executing i_0 is ISA_1 and it is the *post-ISA*

[1] Macro `defstructure` is defined in the ACL2 public book `data-structures/structures`.

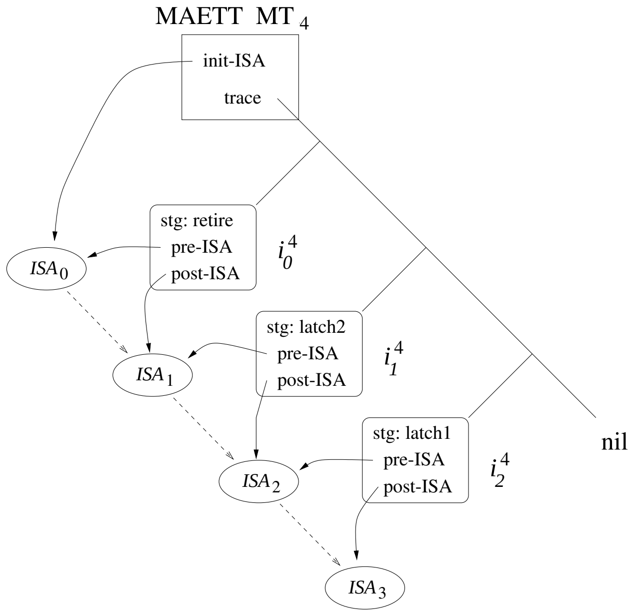

Figure 13.4 The structure of the MAETT intermediate abstraction.

state of i_0. Since it is the state before executing the next instruction, ISA_1 is the *pre-ISA* state of i_1. In this way, the MAETT stores all ISA states that appear during the ISA execution of the program. The dashed lines in the figure show the ISA state transitions.

We can define many values related to an instruction using its INST representation. For example, the expression (`INST-word` i_k^t) defines the instruction word of i_k as the memory value addressed by the program counter in the pre-ISA state of i_k. In the following definition, the expression `(read-mem a mem)` denotes the value of the memory `mem` at address `a`.

```
(defun INST-word (i)
  (read-mem (ISA-pc (INST-pre-ISA i))
            (ISA-mem (INST-pre-ISA i))))
```

From the instruction word, we can calculate the values in the instruction fields *op*, *ra*, *rb*, and *rc*.

```
(defun INST-op (i)
  (op-field (INST-word i)))
(defun INST-ra (i)
  (ra-field (INST-word i)))
(defun INST-rb (i)
  (rb-field (INST-word i)))
```

```
(defun INST-rc (i)
  (rc-field (INST-word i)))
```

We can further define the correct source operand values by reading the source registers in the pre-ISA state. The expression `(read-reg r regs)` denotes the value of register `r` in the register file `regs`.

```
(defun INST-ra-val (i)
  (read-reg (INST-ra i) (ISA-regs (INST-pre-ISA i))))
(defun INST-rb-val (i)
  (read-reg (INST-rb i) (ISA-regs (INST-pre-ISA i))))
```

Finally, we define `INST-result` which calculates the execution result of an instruction. The expression `(ALU-output op src1 src2)` returns the value from the output port of the ALU when the opcode `op`, and source operand values `src1` and `src2` are given to the input ports of the ALU.

```
(defun INST-result (i)
  (ALU-output (INST-op i) (INST-ra-val i) (INST-rb-val i)))
```

These functions are used in the definition of properties that the pipelined machine should satisfy. For example, predicate `INST-latch1-inv` defines the correct intermediate values stored in latch1.

```
(defun inst-latch1-inv (i MA)
  (and (b1p (latch1-valid?  (MA-latch1 MA)))
       (equal (latch1-op (MA-latch1 MA)) (INST-op i))
       (equal (latch1-rc (MA-latch1 MA)) (INST-rc i))
       (equal (latch1-ra (MA-latch1 MA)) (INST-ra i))
       (equal (latch1-rb (MA-latch1 MA)) (INST-rb i)))) .
```

We assume that INST `i` represents an instruction at latch1 in state `MA`. The busy flag `valid?` of latch1 should be 1, because the latch is occupied by the instruction represented by `i`. `(B1p x)` is true if `x` is a bit 1. The opcode of `i`, which has been defined as `(INST-op i)`, should be stored in the `op` field of latch1. Similarly, the predicate checks whether the correct `rc`, `ra`, and `rb` register designators are stored in the corresponding fields of latch1. Another predicate `inst-latch2-inv` defines the correct intermediate values for latch2.

Using these functions, we define the predicate `(inst-invariant i MA)`, which is true if and only if the intermediate values for instruction `i` are correct in state `MA`, regardless of the stage of `i`. We define `(MT-inst-invariant MT MA)` as a predicate that checks that every INST `i` recorded in the `trace` field of MAETT `MT` satisfies the condition `(inst-invariant i MA)`. Intuitively speaking, `(MT-inst-invariant MT MA)` checks that all pipeline intermediate values are correct.

```
(defun inst-invariant (i MA)
  (cond ((equal (INST-stg i) 'latch1)
         (inst-latch1-inv i MA))
        ((equal (INST-stg i) 'latch2)
         (inst-latch2-inv i MA))
        (t t)))
(defun trace-inst-invariant (trace MA)
  (if (endp trace) t
      (and (inst-invariant (car trace) MA)
           (trace-inst-invariant (cdr trace) MA))))
(defun MT-inst-invariant (MT MA)
  (trace-inst-invariant (MT-trace MT) MA))
```

Another property `(regs-match-p MT MA)` is true if and only if the register file in state `MA` is correct, that is, the results of all completed instructions are stored in the register file. In other words, the register file appears as if it were in the post-ISA state of the last completed instruction. With the example given in Table 13.1, the register file state at time 5 should be the same as that in the post-ISA state of i_1. This ideal register file state is calculated from the MAETT with the function `MT-regs`.

```
(defun trace-regs (trace ISA)
  (if (endp trace)
      (ISA-regs ISA)
      (if (not (equal (INST-stg (car trace)) 'retire))
          (ISA-regs ISA)
          (trace-regs (cdr trace)
                      (INST-post-ISA (car trace))))))
(defun MT-regs (MT)
  (trace-regs (MT-trace MT) (MT-init-ISA MT)))
(defun regs-match-p (MT MA)
  (equal (MT-regs MT) (MA-regs MA)))
```

In analogy with `(MT-INST-invariant MT MA)` and `(regs-match-p MT MA)`, we define other properties of our pipelined machine as predicates of the machine state and its MAETT. The following predicate `invariant` is the conjunction of such properties.

```
(defun invariant (MT MA)
  (and (pc-match-p MT MA)
       (regs-match-p MT MA)
       (mem-match-p MT MA)
```

```
        (ISA-chain-p MT)
        (MT-inst-invariant MT MA)
        (MT-contains-all-insts MT MA)
        (MT-in-order-p MT)))
```

In order to introduce the following two theorems, we need three additional functions. `(Flushed? MA)` returns 1 if and only if `MA` is a pipeline flushed state. `(Init-MT MA)` defines the MAETT for any pipeline flushed state `MA`. `(MT-step MT MA sig)` defines the MAETT for the next MA state given that `MA` is the current MA state and `MT` is its MAETT.

```
(defthm invariant-init-MT
    (implies (and (MA-state-p MA) (b1p (flushed? MA)))
             (invariant (init-MT MA) MA)))
(defthm invariant-step
    (implies (and (invariant MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-sig-p sig))
             (invariant (MT-step MT MA sig)
                        (MA-step MA sig))))
```

Theorem `invariant-init-MT` states that every pipeline flushed state satisfies `invariant`. Theorem `invariant-step` states that, if `invariant` is true for the current state, it is also true for the next state. These two theorems show that property `invariant` is an *invariant* condition, and all machine states reachable from a pipeline flushed state satisfy it.

The two theorems above are proven by the divide-and-conquer strategy. Since the predicate `invariant` is a conjunction of several properties, we can establish each condition separately and combine the results to the two theorems. In this sense, the invariant condition can be verified incrementally in our approach. This is particularly useful in verifying a large pipelined machine design, because each condition can be often established by analyzing a part of the pipelined machine.

Since the invariant theorem `invariant-step` involves only `MA-step` but not `ISA-step`, it may appear the theorem is only about the MA behavior. However, the intermediate abstraction MAETT records the ideal ISA execution, this theorem in depth relates the MA and the ISA behavior. In essence, the verified invariant implies that each component of the MA is working correctly with respect to the ISA specification. In the next section, we prove our commutative diagram assuming that the invariant has been already verified.

3. PROVING THE COMMUTATIVE DIAGRAM

First we introduce the theorem that we would like to prove. The following theorem is the formal statement of the commutative diagram discussed earlier.

```
(defthm correctness
  (implies (and (MA-state-p MA)
                (MA-sig-listp sig-list)
                (<= n (len sig-list))
                (b1p (flushed?  MA))
                (b1p (flushed?  (MA-stepn MA sig-list n))))
           (equal (proj (MA-stepn MA sig-list n))
                  (ISA-stepn (proj MA)
                             (num-insts MA sig-list n)))))
```

In this theorem, `MA` is the initial state from which the MA execution starts, and it corresponds to MA_0 in Fig 13.3. We consider the execution of `n` steps with the list of input signals `sig-list`. The length of `sig-list` should be larger than or equal to `n`. The result of `n`-step execution is given as `(MA-stepn MA sig-list n)`, which corresponds to MA_n in the figure. Suppose the initial state `MA` and the final state `(MA-stepn MA sig-list n)` are both pipeline flushed states. The equality in the conclusion compares the two paths of the commutative diagram. The left-hand side runs the MA machine for `n` steps and projects the result to the final ISA state. The right-hand side first projects `MA` to the initial ISA state `(proj MA)` and then runs the ISA machine for `(num-insts MA sig-list n)` steps. The function `num-insts` returns the number of instructions executed in the `n`-step MA execution, which is given as m in Fig. 13.3.

One question is how the function `num-insts` counts the number of instructions executed during the `n`-step MA execution. The expression `(num-insts MA sig-list n)` first constructs the MAETT for the final MA state MA_n. This MAETT is a complete history of instructions executed during the `n`-step MA execution. The function `num-insts` simply counts the number of the instructions recorded in this MAETT.

```
(defun MT-num-insts (MT) (len (MT-trace MT)))
(defun num-insts (MA sig-list n)
  (MT-num-insts (MT-stepn (init-MT MA) MA sig-list n)))
```

A proof sketch of theorem `correctness` follows. Component by component, we show the equality in the conclusion of the theorem. There are three components to compare: the program counter, the register file, and the memory. We will discuss the equality with respect to the register file in detail. Equalities for other components are proven similarly.

To ease the following arguments, we use the symbols shown in Fig. 13.3. The left-hand side of the conclusion of the `correctness` theorem is given as `(proj` MA_n`)`. Since the initial ISA state `(proj MA)` is ISA_0 in the figure, the right-hand side is given as `(ISA-stepn` ISA_0 m`)`. We need to prove the following equality for the register file:

$$\texttt{(ISA-regs (proj } MA_n\texttt{))} = \texttt{(ISA-regs (ISA-stepn } ISA_0\ m\texttt{))}. \tag{13.1}$$

Let us assume that MT_n represents the MAETT for state MA_n. From the two lemmas `invariant-init-MT` and `invariant-step`, `(invariant` MA_n MT_n`)` is true. From the definition of `invariant`, the property `(regs-match-p` MT_n MA_n`)` is derived. The definition of `regs-match-p` implies that the final register file state `(MA-regs` MA_n`)` is equal to the ideal register file state `(MT-regs` MT_n`)`. Using the definition of `proj`,

$$\begin{aligned} & \texttt{(ISA-regs (proj } MA_n\texttt{))} \\ = & \texttt{(MA-regs } MA_n\texttt{)} \\ = & \texttt{(MT-regs } MT_n\texttt{)}\ . \end{aligned}$$

Let $(i_0^n \ldots i_{m-1}^n)$ be the list of instructions in the `trace` field of MT_n. Because the final state MA_n is flushed, the execution of all instructions in this list are completed and `(INST-stg` i_k^n`)` = `'retire` for all k such that $0 \le k < m$. Hence, `(MT-regs` MT_n`)` = `(ISA-regs (INST-post-ISA` i_{m-1}^n`))` because `(MT-regs` MT_n`)` returns the register file in the post-ISA state of the last completed instruction, which is represented by i_{m-1}^n. Since the post-ISA state of i_{m-1}^n is the state that results from executing m instructions i_0 through i_{m-1} by the ISA, it is equal to `(ISA-stepn` ISA_0 m`)`. Therefore,

$$\begin{aligned} & \texttt{(MT-regs } MT_n\texttt{)} \\ = & \texttt{(ISA-regs (INST-post-ISA } i_{m-1}^n\texttt{))} \\ = & \texttt{(ISA-regs (ISA-stepn } ISA_0\ m\texttt{))}\ . \end{aligned}$$

From the equalities shown above, we derive Equation (13.1) and conclude the proof of the commutative diagram with respect to the register file.

Since the proof of our invariant discussed in the previous section has already shown that each MA component, such as the resister file, is in the correct state with respect to the ideal ISA execution, deriving the commutative diagram from the invariant condition is not hard. In a very simplified picture, our proof deduces by induction the commutative diagram, which involves n MA state transitions, from the invariant condition, whose verification involves only the current and the next state.

The `correctness` theorem is vacuous if the MA never reaches a pipeline flushed state, because the last hypothesis of the theorem does not hold. However, the following theorem proves that we can flush the pipelined machine by running the MA model long enough without fetching new instructions. The input signal to the pipelined machine controls instruction fetching, and the machine does not fetch a new instruction when the input is 0.

```
(defthm liveness
  (implies (MA-state-p MA)
           (b1p (flushed?
                  (MA-stepn MA
                            (zeros (flush-cycles MA))
                            (flush-cycles MA)))))) .
```

The expression `(zeros n)` returns a list of 0's whose length is n. The *witness function* `flush-cycles` returns the number of steps necessary to flush out all instructions in the pipeline, proving the existence of such a number.

Even though the 3-stage pipelined machine verified here is simple, our verification approach can be scaled to a more complex pipelined machine model. The author has used a similar approach to verify a microprocessor model which issues and completes instructions out-of-order, executes instructions speculatively, and implements interrupts[Sawada and Hunt, 1998, Sawada, 1999]. To verify such a processor model, we had to extend the MAETT to record more information about instructions. Also we needed to verify more complex invariant than the 3-stage pipelined machine. However, the general approach to the problem does not change.

Exercise 13.1 *The expression* `(INST-in-order i j MT)` *is true if and only if the instruction represented by* `i` *precedes the one represented by* `j` *in program order. Using the MAETT representation, we can define it as follows, because the MAETT records instructions in program order in a list.*

```
(defun member-in-order (i j lst)
  (member-equal j (cdr (member-equal i lst))))
(defun INST-in-order-p (i j MT)
  (member-in-order i j (MT-trace lst)))
```

We want to prove that `(INST-in-order-p i j MT)` *defines a partial order between* `i` *and* `j` *given that* `MT` *is fixed. This requires to prove the asymmetry and transitivity of* `member-in-order` *with the following theorems.*

```
(defthm  member-in-order-asymmetry
    (implies (and (member-in-order i j lst))
```

```
             (not (member-in-order j i lst))))
(defthm member-in-order-transitivity
    (implies (and (member-in-order i j lst)
                  (member-in-order j k lst))
             (member-in-order i k lst)))
```

Prove them. If not provable, why? What kind of additional hypotheses allow the theorems to hold?

References

[Sawada, 1999] Sawada, J. (1999). *Formal Verification of an Advanced Pipelined Machine.* PhD thesis, University of Texas at Austin. `http://www.cs.utexas.edu/users/sawada/dissertation/`.

[Sawada and Hunt, 1997] Sawada, J. and Hunt, Jr., W. A. (1997). Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer Verlag.

[Sawada and Hunt, 1998] Sawada, J. and Hunt, Jr., W. A. (1998). Processor verification with precise exceptions and speculative execution. In Hu, A. J. and Vardi, M. Y., editors, *computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer Verlag.

Index